Logic Programming and AI

Introduction to Prolog

Prolog Labs

- Lab assignment: see LPA web page.
- No hand-in of lab problem solutions.
- “Official” solutions will be posted on the web.

Prolog Interpreter

- We will be using SWI Prolog, a freeware interpreter: http://www.swi-prolog.org
- There is a commercial Prolog interpreter on the computer science servers: Sicstus Prolog.

Prolog books

- Bratko: *Prolog Programming for Artificial Intelligence* (3rd edition)
- Sterling & Shapiro: *The Art of Prolog* (2nd edition).
- Rob Lucas: *Mastering Prolog*
- Clocksim & Mellish: *Programming in Prolog: Using the ISO standard* (5th edition)

Free Prolog Books

- Nilsson & Maluszynski: *Logic, Programming, and Prolog* (http://www.ida.liu.se/~ulfni/lpp)
- Blackburn, Bos & Striegnitz: *Learn Prolog Now*

  (http://www.coli.uni-sb.de/~kris/learn-prolog-now/)

Logic Programming

- Declarative programming paradigm: emphasis on **what** the program does, not **how** it goes about doing it.
- Use the language of logic to specify a problem, and then use a proof procedure to do the computation.
- Prolog: use SLD-resolution proof procedure.

Logic Programming System

- A Logic Programming System should be such that if a solution to our problem exists, then a proof should exist given the problem specification.
- **Program**: logical specification of a problem.
- **Computation:** proof (i.e. running the program).

Advantages

- Very high level language
- Easy to prototype
- Shorter programs
- More readable programs
- Ideal for many AI applications

Disadvantages

- Not intuitive to procedural programmers.
- Steep learning curve initially.
- Program execution comparatively slow: often prototyping in Prolog, and then optimized implementation in C/C++

Prolog without Variables

Consider the following program:

```
ed_lives_in_york.

ed_is_a_cs_student.

ed_does_lpa.
```

Once we load this program into Prolog, we can type in the following query:

```
?- ed_lives_in_york.
```

and Prolog will respond with: `Yes`

Conjunction

We could ask the following query from Prolog: is it the case that Ed lives in York and that he does LPA?

This is how:

```
-? ed_lives_in_york, ed_does_lpa.
```

The obvious reply from Prolog given our previous program is:

```
Yes
```

Adding Simple Rules

Consider this program:

```
ed_lives_in_york.

ed_is_a_cs_student.

ed_does_lpa.

ed_loves_logic :- ed_is_a_cs_student,
                  ed_does_lpa.
```

The rule above is equivalent in logic to

- *ed_loves_logic* $\leftarrow$ *ed_is_a_cs_student* $\wedge$ *ed_does_lpa.*
- *ed_is_a_cs_student* $\wedge$ *ed_does_lpa* $\rightarrow$ *ed_loves_logic.*

The Logic Perspective

Ed's program in logic

1. *ed_lives_in_york*

2. *ed_is_a_cs_student*

3. *ed_does_lpa*

4. $ed_is_a_cs_student \wedge ed_does_lpa \rightarrow ed_loves_logic$

Can we deduce *ed_loves_logic* from the premises above?

What response can we expect for the query:

```
-? ed_loves_logic.
```

Another Example (1a)

```
p.

s.

q :- p, r.

r :- s, p.
```

Can we deduce `q` from this program?

Another Example (1b)

How does Prolog go about deducing q? SLD!

1. to prove q, we must prove both p and r
2. p is true (given)
3. to prove r, we must prove both s and p
4. both s and p are true (given)
5. done!

This way, by working through the implications, Prolog only has to prove things that are relevant in proving the goal.

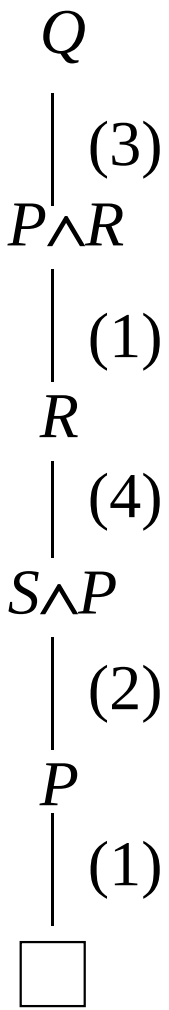
Another Example (2a)

1. P
2. S
3. $P \wedge R \rightarrow Q$
4. $S \wedge P \rightarrow R$

Q

Is this a valid argument of propositional logic?

Another Example (2b)



Prolog Body Parts

- Informally, the thing before the :- is the *head* and the thing after the :- is the *body*.
- In Prolog, we can only have a head with *one* predicate, whereas the body can be a conjunction or disjunction of predicates.
- Things that end in a full-stop are known as *clauses*.

Disjunction

Disjunction exists in Prolog as the semicolon "`;`" but it is good practice to avoid using it.

In fact, we can always get rid of disjunctions from the body of a clause as follows:

```
p :- q ; (r,s).
```

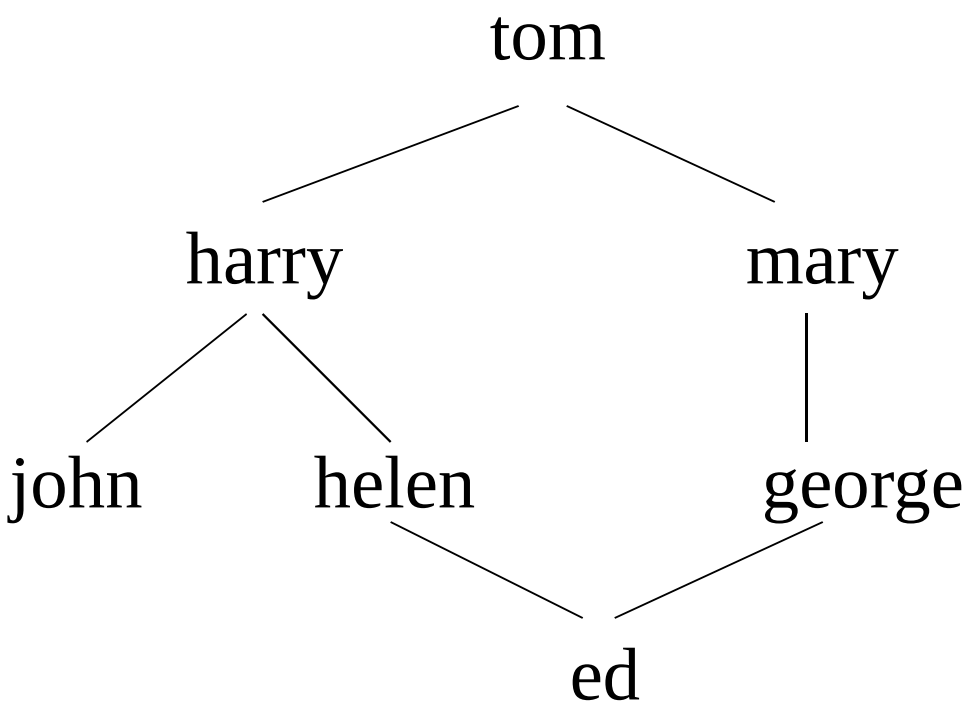
is the same as :

```
p :- q.
p :- r,s.
```

Beyond Propositional

- So far, we have only considered ground clauses and queries.
- Restriction to propositional logic makes it infeasible to handle bigger databases.
- The real power of Prolog is only revealed with the use of variables.

The parent database (1)



Well, it could have happened in ancient Greece ….

Parent Program

```
parent(tom, harry).
parent(tom, mary).
parent(harry, john).
parent(harry, helen).
parent(mary, helen).
parent(mary, george).
parent(helen, ed).
parent(george, ed).
```

Queries (1)

How can we find out who is the parent of whom?

For example, what query would we give to Prolog to find the parents of Ed?

```
?- parent(X, ed).
```

Prolog would respond with:

```
?- X = helen
```

and it would sit there and wait. . .

Queries (2)

If we press the semicolon (which means *give me another answer*), Prolog would respond with:

```
?- parent(X, ed).

X = helen ;

X = george
```

and if we tried the semicolon again, we'd get:

```
No
```

Queries (3)

What if we wanted to know every parent-child pair?

```
?- parent(X,Y).
X = tom
Y = harry ;
X = tom
Y = mary ;
X = harry
Y = john ;
```

and so on until pressing the semicolon would return `No`

Prolog and Resolution

The query

```
?- parent(X, ed).
```

is equivalent to ∃X: *parent*(*X*, *ed*) in the FOPC.

Can we have more complex queries?

Sure, for example, the following will return Ed's grandparents:

```
?- parent(X, ed), parent(Y,X).
```

How about a great grandfather?

The grandparent Relation

Add the following rule to the database:

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

The same in logic:

$\forall X \, \forall Y \, \forall Z \, (grandparent(X,Y) \leftarrow parent(X,Z) \wedge parent(Z,Y))$

The same in English: If some individual *X* is the parent of *Z* and *Z* is the parent of some other individual *Y,* then *X* is the grandparent of *Y.*

A New Program

```
parent(tom, harry).
parent(tom, mary).
parent(harry, john).
parent(harry, helen).
parent(mary, helen).
parent(mary, george).
parent(helen, ed).
parent(george, ed).
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

Queries on Grandparents

A query to find Ed's grandparents:

```
?- grandparent(X,ed).
```

will return Harry and Mary.

A query to find Tom's grandchildren:

```
?- grandparent(tom,X).
```

will return John, Helen and George.

Prolog vs. C (1)

A bunch of queries:

1. given a person, compute his/her grandchildren
2. given a person, compute his/her grandparents
3. given two people, compute whether the former is the grandparent of the latter
4. compute all grandparent/grandchildren pairs

Prolog vs. C (2)

The same relation, that of parenthood, is involved in all cases. In Prolog, we can get away by just defining just one relation:

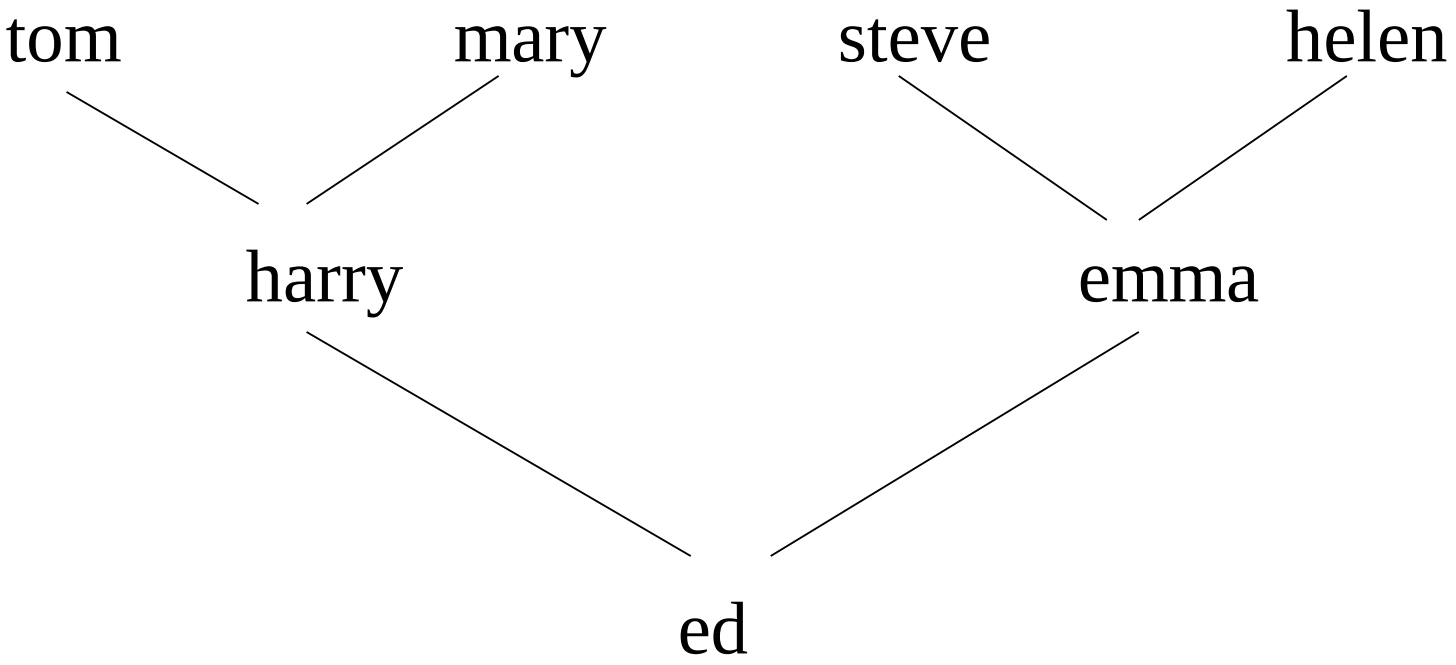
```
grandparent(X,Y) :- parent(X,Z),
                    parent(Z,Y).
```

We then vary the query. The variables X and Y in `grandparent` may be:

instantiated mapped to constants

uninstantiated mapped to variables

A Smaller Parents Database



The Parent Database in Prolog

```
parent(tom, harry).
parent(mary, harry).
parent(steve, emma).
parent(helen, emma).
parent(harry, ed).
parent(emma, ed).
```

Polymodality

Polymodality is the ability to have multiple **modes** in our programs or queries i.e. multiple patterns of instantiation:

```
?- parent(tom, harry).

?- parent(mary, X).

?- parent(X, mary).

?- parent(X,Y).
```

Cleaner Output

We want to find Tom's children who have children of their own:

```
?- parent(tom, X), parent(X,Y).

X = harry

Y = ed ;

No
```

But, we're just interested in the value of X, not Y , as long as some Y exists.

Anonymous Variables

We can use **anonymous** variables:

```
?- parent(tom, X), parent(X,_).
X = harry ;
No
```

Anonymity Pitfalls

The query:

```
?- parent(tom, _), parent(_,_).
```

is not the same as

```
?- parent(tom, X), parent(X,_).
```

but is, in fact, equivalent to:

```
?- parent(tom, X), parent(Y,Z).
```

Listing

To get a listing of your Prolog program:

```
?- listing.
```

Sometimes, as you change your program, Prolog “remembers” older definition of the predicates in your code. In these cases, it is a good idea to exit the interpreter and restart it to clear the memory.

Trace

```
[trace] ?- parent(tom, X), parent(X,Y).
Call: (8) parent(tom, _G284) ? creep
Exit: (8) parent(tom, harry) ? creep
Call: (8) parent(harry, _G287) ? creep
Exit: (8) parent(harry, ed) ? creep
X = harry
Y = ed ;
Fail: (8) parent(harry, _G287) ? creep
Fail: (8) parent(tom, _G284) ? creep
No
```

Ancestors

- **Question**: Who are my ancestors?
- My ancestors are my parents, my parent's parents, my parents' parents' parents, my parents' parents' parent's parents etc. . .
- My ancestors are either my parents or the ancestors of my parents.
- What is the definition of the `ancestor` relation in Prolog?

Ancestors in Prolog

```
parent(tom, harry).
parent(mary, harry).
parent(steve, emma).
parent(helen, emma).
parent(harry, ed).
parent(emma, ed).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Queries and Search Trees

What is the search tree associated with the query: `ancestor(X,ed)`?

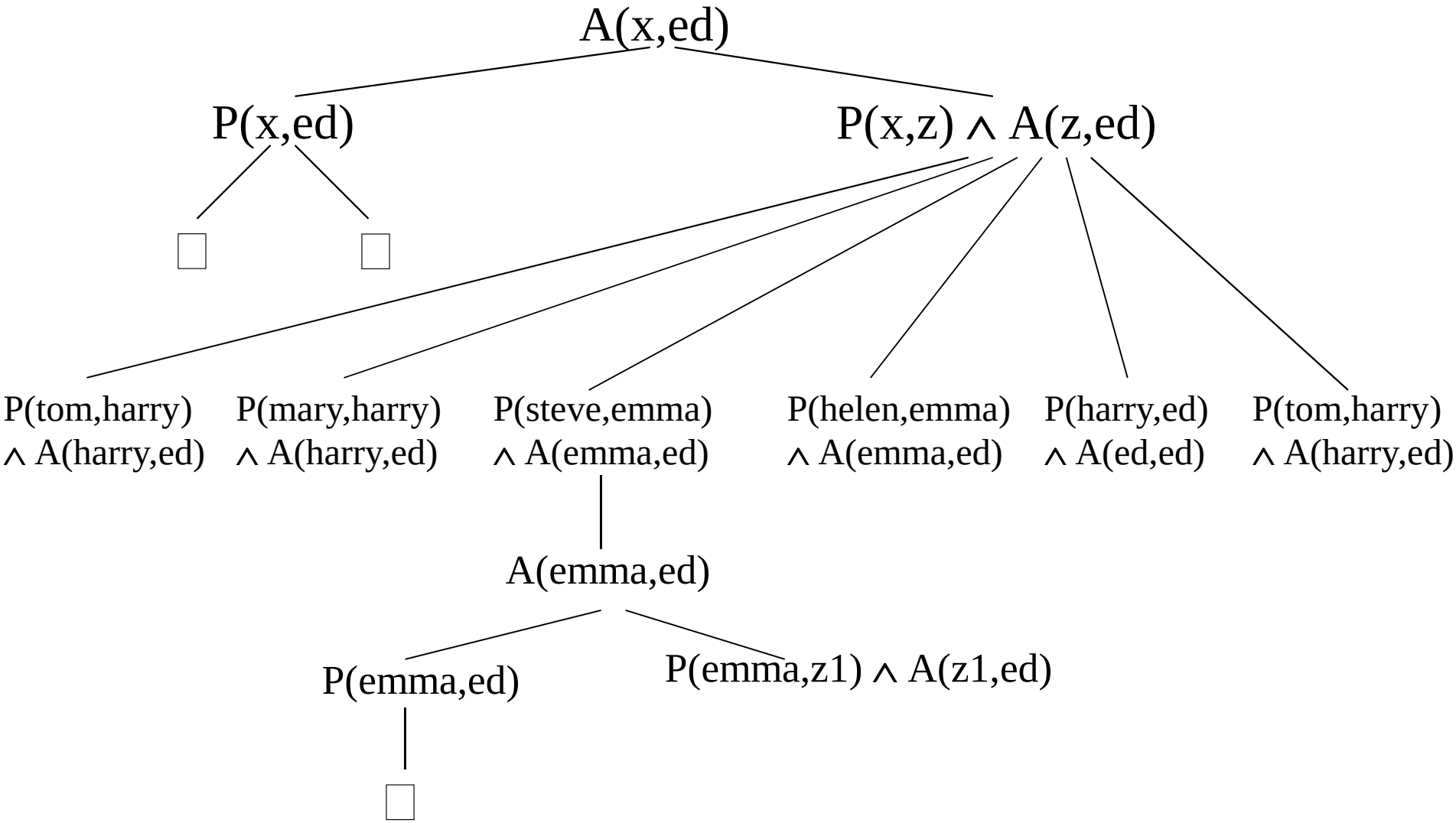
1. Translate the Prolog program back into the FOPC
2. Translate the query to a conclusion for the argument whose premises are the sentences from the Prolog program
3. Do the SLD-resolution proof for the resulting argument!

From Prolog to Logic

1. Parent(tom, harry)
2. Parent (mary, harry)
3. Parent (steve, emma)
4. Parent (helen, emma)
5. Parent (harry, ed)
6. Parent (emma, ed)
7. Parent (x, y) $\rightarrow$ Ancestor(x, y)
8. Parent (x, z) $\wedge$ Ancestor (z, y) $\rightarrow$ Ancestor (x, y)

Ancestor (x, ed)

Partial Search Space



Prolog's Search Strategy

- Branches in the previous tree represent alternative solutions.
- Prolog searches the tree in a depth-first manner.
- Solutions are found by the Prolog interpreter in the same order as they are discovered by DFS.
- The order of the clauses in the Prolog program is important.

A Different Program

```
parent(tom, harry).
parent(mary, harry).
parent(steve, emma).
parent(helen, emma).
parent(harry, ed).
parent(emma, ed).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
ancestor(X,Y) :- parent(X,Y).
```

What's the difference?

- What is the search tree associated with the same query as before: `ancestor(X,ed)`?
- Is the search space different?
- Are the same solutions found?

Another Pitfall

Consider the following program:

```
likes(ed,emma).

likes(ed,john).

likes(ed,X) :- likes(X,ed).
```

Another Pitfall (2)

What is the search space associated with the query:

```
likes(ed,X).
```

```
?- likes(ed,X).
```

```
X = emma ;
```

```
X = john ;
```

What happens next?

Another Pitfall (3)

Prolog went into an infinite loop trying to prove the goal: `likes(ed,ed).`

Infinite Loops

- We can expect infinite computation when using infinite relations (or mutually recursive ones).
- But, the way Prolog does things means that even some finite queries can fail!
- Prolog's strategy:
  - Try clauses from top to bottom.
  - Try subgoals from left to right.

Logic vs. Prolog

Four logically equivalent programs:

```
a(X,Y) :- p(X,Y).
a(X,Y) :- p(X,Z), a(Z,Y).
```

```
a(X,Y) :- p(X,Z), a(Z,Y).
a(X,Y) :- p(X,Y).
```

```
a(X,Y) :- p(X,Y).
a(X,Y) :- a(Z,Y), p(X,Z).
```

```
a(X,Y) :- a(Z,Y), p(X,Z).
a(X,Y) :- p(X,Y).
```

Rules of Thumb

- Do not have recursive calls as the first subgoal (so-called *left recursion*).
- Put the base case as the first clause.
- Be prepared to break these rules.

Data Structures

- Prolog supports only the ***term*** data structure: A term in Prolog is a ***constant*** or a ***variable*** or a ***function symbol*** of arity n applied to n terms.
- Sometimes, the words ***function symbol*** and ***functor*** are used interchangeably.
- Variables can have numerical or list values (see later lectures).

LPA staff database

Consider the following database:

```
person(name(matthew, grounds),
       sex(male)).
person(name(alan, frisch),
       sex(male)).
person(name(daniel, kudenko),
       sex(male)).
```

Is any one of them called matthew?

```
?- person(name(matthew, _), sex(_)).

Yes
```

An Alternative

```
person(firstname(matthew),
       surname(grounds),
           sex(male)).

person(firstname(alan),
       surname(frisch),
       sex(male)).

person(firstname(daniel),
       surname(kudenko),
       sex(male)).
```

Lists

A list is an ordered set of elements. In Prolog, lists are used very extensively so it's good to understand them early.

In Prolog:

- a list is either empty `[]`
- or it is made up of a head `H` and a tail (body) `T` as follows: `[H|T]`

Examples of Lists

`[1,2,3,4,5]` is a list with head `1` and tail `[2,3,4,5]`.

`[likes(ed,guiness), likes(ed,ghost)]` is a list with head `likes(ed,guiness)` and tail `[likes(ed,ghost)]`

Syntax of Lists

list → `[ ]` | `.(`head`,`tail`)`

head → term

tail → list

Note that in the definition above, `.(head,tail)` is Prolog's way of writing `[head | tail]`

More Syntactic Sugar

The list: `[X1, X2, X3 | X4]` is a list:

1. First element is `X1`
2. Second element is `X2`
3. Third element is `X3`
4. Remainder is the list `X4`

Note that the list above is different to:
`[X1, X2, X3, X4]`

List Notation

Commas “,” separate individual list elements. The bar “|” separates an element and the rest of the list.

1. `[]` contains zero elements
2. `[Y|Ys]` contains at least one element
3. `[Y1,Y2|Ys]` contains at least two elements
4. `[Y1|Y2,Y3]` is a syntax error
5. `[Y1 | [Y2,Y3]]` is the same as `[Y1,Y2,Y3]`
6. `[Y1 | [Y2 | Y3]]` is the same as
   `[Y1,Y2 | Y3]`

Pure Prolog Syntax (1)

Program → Clause. |
Clause. Program

Clause → Fact |
HornClause

Fact → Literal

Literal → p |
p(t1, t2, … , tn)

where p is a predicate symbol and t1, t2, … , tn are terms.

Pure Prolog Syntax (2)

HornClause → Head :- Body

Head → Literal

Body → Literal |
Literal, Body |
Literal; Body

A First "Real" Program

- Write a program that can evaluate Boolean expressions.
- What do we need to write such a program?
  - Decide which Boolean functions to include ('and', 'or', 'not', etc).
  - How do we define the Boolean functions in Prolog?
  - Choose a value for True and one for False.
  - Back to truth tables!

Boolean Expressions

- Represent the ‘and’, ‘or’ and ‘not’ Boolean functions.
- Represent *TRUE* by the constant `true`.
- Represent *FALSE* by the constant `false`.
- Note that a truth table is simply a relation.
- Represent the basic truth tables with ground rules.
- Implement the evaluation mechanism through Prolog rules.

Truth Tables

P	¬P
T	F
F	T

P	Q	P ∧ Q
T	T	T
T	F	F
F	T	F
F	F	F

P	Q	P ∨ Q
T	T	T
T	F	T
F	T	T
F	F	F

Ground Clauses

```
eval(and(true, true), true).
eval(and(false, true), false).
eval(and(true, false), false).
eval(and(false, false), false).

eval(or(true, true), true).
eval(or(true, false), true).
eval(or(false, true), true).
eval(or(false, false), false).

eval(not(true), false).
eval(not(false), true).
```

Boolean Evaluation (1)

First, we need some way to encapsulate the Boolean functions so that Prolog can match them:

```
eval(v(true), true).

eval(v(false), false).
```

Finally, we are ready to ask ourselves: What does it mean for P $\wedge$ Q, P $\vee$ Q, $\neg$P to be true?

Boolean Evaluation (2)

```
eval(and(A, B), V):-
     eval(A, AV),
     eval(B, BV),
     eval(and(AV, BV), V).

eval(or(A, B), V):-
     eval(A, AV),
     eval(B, BV),
     eval(or(AV, BV), V).

eval(not(A), V):-
     eval(A, AV),
     eval(not(AV), V).
```

Demonstration (1)

Forward reasoning: what is the truth value of the expression *TRUE* $\wedge$ *FALSE*?

```
?- eval(and(v(true), v(false)), X).

X = false ;

No
```

Demonstration (2)

Backward reasoning: what truth values for X and Y make the expression (X ∨ Y) ∧ (Y ∧ X) true?

```
?- eval(and(or(v(X), v(Y)),
        and(v(Y), v(X))
        ), true).

X = true

Y = true ;

No
```

Unification (1)

Prolog uses resolution as its only rule of inference!
The unification operator is '='

Examples:

```
?- i_love_beer(ed,said) = i_love_beer(X,said).

X = ed


?- hello(you, f(Y)) = hello(X, f(fool)).

X = you
Y = fool
```

Unification (2)

Sometimes, unification can also fail:

```
?- p(X,f(X)) = p(f(X), X).
X = f(f(f(f(f(f(f(f(f(f(...)))))))))
```

```
?- p(X) = p(f(X)).
X = f(f(f(f(f(f(f(f(f(f(...)))))))))
```

Operators in Prolog

Say we want to write that 4 is less than 5 in Prolog. How can we do it?

As it turns out, Prolog does have these operators built-in:

```
?- <(4,5).

Yes

?- >(4,5).

No
```

```
?- 4 < 5.

Yes

?- 4 > 5.

No
```

Infix and Prefix

- The operator/functor "<" in X < Y is an *infix* operator.
- X < Y is internally translated to the term < (X, Y)
- The operator/functor "<" in < (X, Y) is a *prefix* operator.

Mixedfix

- Another Prolog builtin operator is **[_ | _].** This is a *mixedfix* operator which also takes two arguments. This operator is also translated to **.(X,Y)** internally.
- Note that "**.**" is a functor symbol in this context and not a clause terminator.

User-defined Operators

- Prolog also allows you to define your own operators (`op/3`). For example, instead of writing: `and(v(1),v(0))` we can write: `v(1) & v(0)` *provided* we define "`&`" as a new *infix* operator.
- Often, there's no need to define operators. Check the user manual for more details.

Procedural vs. Declarative Semantics (1)

The prolog program:

```
p :- q,r.
```

is “supposed” to mean the same as:

```
p :- r,q.
```

But, in practice, they are different.

Procedural vs. Declarative Semantics (2)

In the program:

```
p :- q,r.
```

how does Prolog solve the query:

```
?- p.
```

1. first solve `q`
2. then solve `r`

Procedural vs. Declarative Semantics (3)

- Unfortunately, we must sometimes take Prolog's way of solving queries into account:
  - to avoid infinite search
  - for efficiency
  - to *hack* certain tricks

Prolog Tips and Tricks (1)

- Spell the predicates correctly.
- Pass the right number of arguments to each function.
- Locate operators and determine their precedence.
- Introduce brackets to avoid confusion.
- Determine the pattern of instantiation of variables and make sure it matches the intended use by the rest of the program.

Prolog Tips and Tricks (2)

- It is good practice to:
  - put all ground clauses of a predicate at the top of the Prolog file.
  - put all other clauses that define a predicate together below the ground ones
- Common errors:
  - Forgetting the full stop at the end of a clause
  - Having unmatched round brackets ( ), square Brackets [ ] or curly brackets (braces) { }

Trace

A very useful tool that comes with Prolog is the `trace` command. It has the following *events*:

1. `Call`: Prolog starts trying to solve a goal.
2. `Exit`: Some goal has been solved by Prolog.
3. `Redo`: Prolog begins to re-solve a goal.
4. `Fail`: Prolog has failed to solve a goal.

Trace: Spy Points

- To trace only a specific predicate `pred` of your program, set a *spy point*: `spy(pred).`
- To remove the spypoint for a predicate `pred`: `nospy(pred).`
- If more than one version of `pred` exists (e.g., one of arity 2 and one of arity 3), we can spy on them individually or together. Example: `spy(pred/2).`

Trace: Leashing

- Another useful feature is *leashing* which controls which parts of a trace the user is shown.
- `leash(+call)` tells Prolog that we are interested in the call event.
- `leash(-fail)` tells Prolog that we are not interested in the fail event.
- `leash([+call,+exit,-redo,-fail]).` tells Prolog that we are interested in calls and exits but not in redos and failures.

Trace Commands (1)

- `creep` (the “Return” key) tells Prolog to carry on until the next leashed event.
- `skip` (the “s” key) tells Prolog to carry on and produce no trace messages until another event occurs involving the current goal.
- `leap` (the “l” key) tells Prolog to carry on and produce no trace messages until either a spy point is reached or another event occurs involving the current goal.

Trace Commands (2)

- `break` (the “b” key) tells Prolog to exit the current trace. It gives us a temporary new interpreter (shell) and we can resume our trace with “Ctrl D”.
- `abort` (the “a” key) tells Prolog to abandon the current trace altogether.
- `exit` (the “c” key) tells Prolog to exit the current trace and terminate the Prolog interpreter thus returning us to the shell.

SWI Manual

- As with all programming languages, the most important aid in learning to develop good programs with Prolog is the *language manual*.
- Get it at: `http://www.swi-prolog.org`
- The Prolog interpreter also has the `help/0` and `help/1` predicates for you to use which access the manual.

Back to Lists

Consider the following list-membership program:

```
memb(X,[X|_]).
memb(X,[_|T]) :- memb(X,T).
```

Note that `memb` is not a fully polymodal predicate as we can't ask Prolog the queries:

```
?- memb(1, X).
?- memb(X,Y).
```

Types (1)

Prolog is not a typed language, e.g. the following queries perhaps should give type errors, but they don't:

```
?- memb(a, [a|b]).

Yes

?- memb(b, [a|b]).

No
```

How does Prolog come to the above two conclusions?

Types (2)

Suppose we define a predicate `list/1`, where `list(X)` would be true if `X` is a list:

```
list([]).

list([X|Xs]) :- list(Xs).
```

How could we then make the `memb/2` predicate more type safe?

Types (3)

Then, we *could* make `memb` type safe(r):

```
memb(X, [X | Xs]) :- list(Xs).
memb(X, [Y | Ys]) :- memb(X, Ys).
```

Now, the previous queries behave as expected:

```
?- memb(a, [a | b]).

No

?- memb(b, [a | b]).

No
```

Appending Two Lists (1)

We'd like to define the predicate `append/3` where `append(Xs, Ys, Zs)` is true if `Zs` is the list formed by concatenating `Ys` to the end of `Xs`.

For example:

```
?- append([a, b, c], [d, e], L).
```

```
L = [a, b, c, d, e]
```

Appending Two Lists (2)

The base case of `append/3` concerns the case where `Xs` is the empty list:

```
append([], Ys, Ys).
```

In English: if you try to append the empty list to another list, simply return the other list.

Appending Two Lists (3)

If the first argument of `append/3` is a non-empty list (i.e. one that has at least one element):

```
append([X|Xs], Ys, [X|Zs]) :-
    append(Xs, Ys, Zs).
```

In English: if you try to append a list with at least one element in it to a second list, return a new list whose head is the head of the first list and whose tail is the second list glued on to the tail of the first list.

Appending Two Lists (4)

- We recurse on only one argument, as is *often* the case.
- `append` is left-recursive.
- `append` describes an infinite relation.
- `append` is also not fully polymodal.
- But: some of the modes that `append` offers are very useful indeed.

Reversing a List

An example of the usefulness of `append` is reversing a list:

```
reverse([], []).

reverse([X | Xs], Ys) :-
    reverse(Xs, Zs),
    append(Zs, [X], Ys).
```

You'll get lots more practice in the practicals!